
Computer Systems for Data Science

Topic 6

MapReduce
Spark
Streaming



Borrowed from Jeff Ullman, Cristiana Amza and Indranil Gupta

MapReduce



MapReduce and Hadoop

- SQL and ACID are a very useful set of abstractions
- But: quite heavy, hard to scale
- MapReduce is a more limited style of programming designed for:
 1. Easy parallel programming.
 2. Invisible management of hardware and software failures.
 3. Easy management of very-large-scale data.
- It has several implementations, including Hadoop, Flink, and the original Google implementation just called “MapReduce.”
- It is also used in Spark

MapReduce in a Nutshell

- A MapReduce job starts with a collection of input elements of a single type.
 - Technically, all types are key-value pairs.
- Apply a user-written **Map function** to each input element, in parallel.
 - **Mapper** applies the Map function to a single element.
 - Many mappers grouped in a **Map task** (the unit of parallelism).
- The output of the Map function is a set of 0, 1, or more *key-value pairs*.
- The system sorts all the key-value pairs by key, forming key-(list of values) pairs.

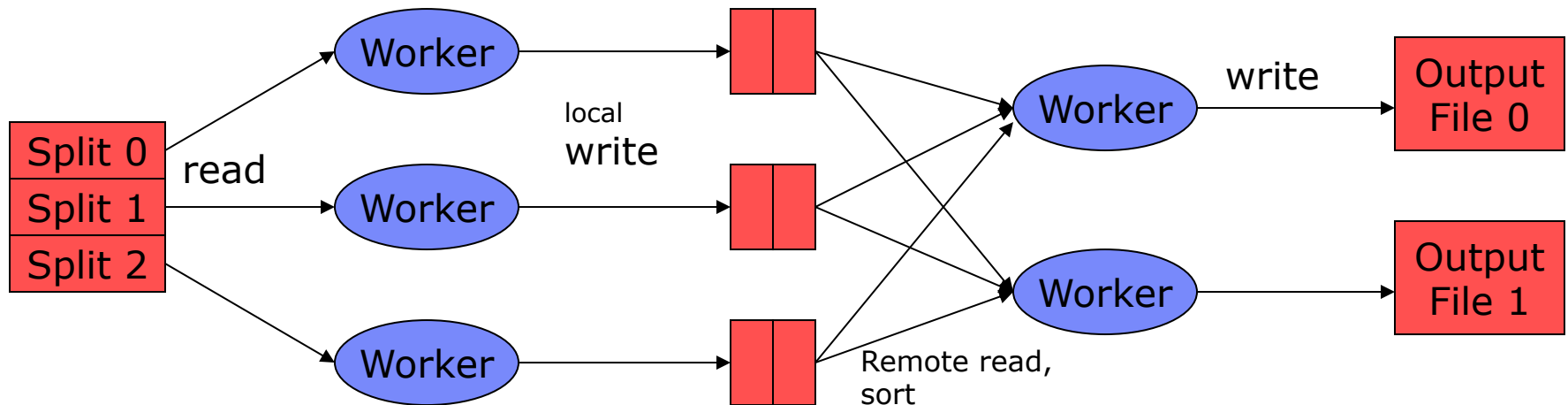
In a Nutshell – (2)

- Another user-written function, the **Reduce function**, is applied to each key-(list of values).
 - Application of the Reduce function to one key and its list of values is a **reducer**.
 - Often, many reducers are grouped into a **Reduce task**.
- Each reducer produces some output, and the output of the entire job is the union of what is produced by each reducer.

MapReduce workflow

Input Data

Output Data



Map

extract something you
care about from each
record

Reduce

aggregate,
summarize, filter,
or transform

Example: Word Count

- We have a large file of documents (the input elements).
- Documents are words separated by whitespace.
- Count the number of times each distinct word appears in the file.

Word Count Using MapReduce

```
map(key, value):
```

```
// key: document ID; value: text of document
```

```
  FOR (each word w IN value)
```

```
    emit(w, 1);
```

```
reduce(key, value-list):
```

```
// key: a word; value-list: a list of integers
```


```
  result = 0;
```

```
  FOR (each integer v on value-list)
```

```
    result += v;
```

```
  emit(key, result);
```

Expect to be all 1's, but
"combiners" allow local
summing of
integers with the same
key before passing to
reducers.



Mapper

- Reads in input pair `<Key, Value>`
- Outputs a pair `<K', V'>`
 - Let's count number of each word in user queries (or Tweets/Blogs)
 - The input to the mapper will be `<queryID, QueryText>`:
`<Q1, "The teacher went to the store. The store was closed; the store opens in the morning. The store opens at 9am." >`
 - The output would be:
`<The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store, 1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>`

Reducer

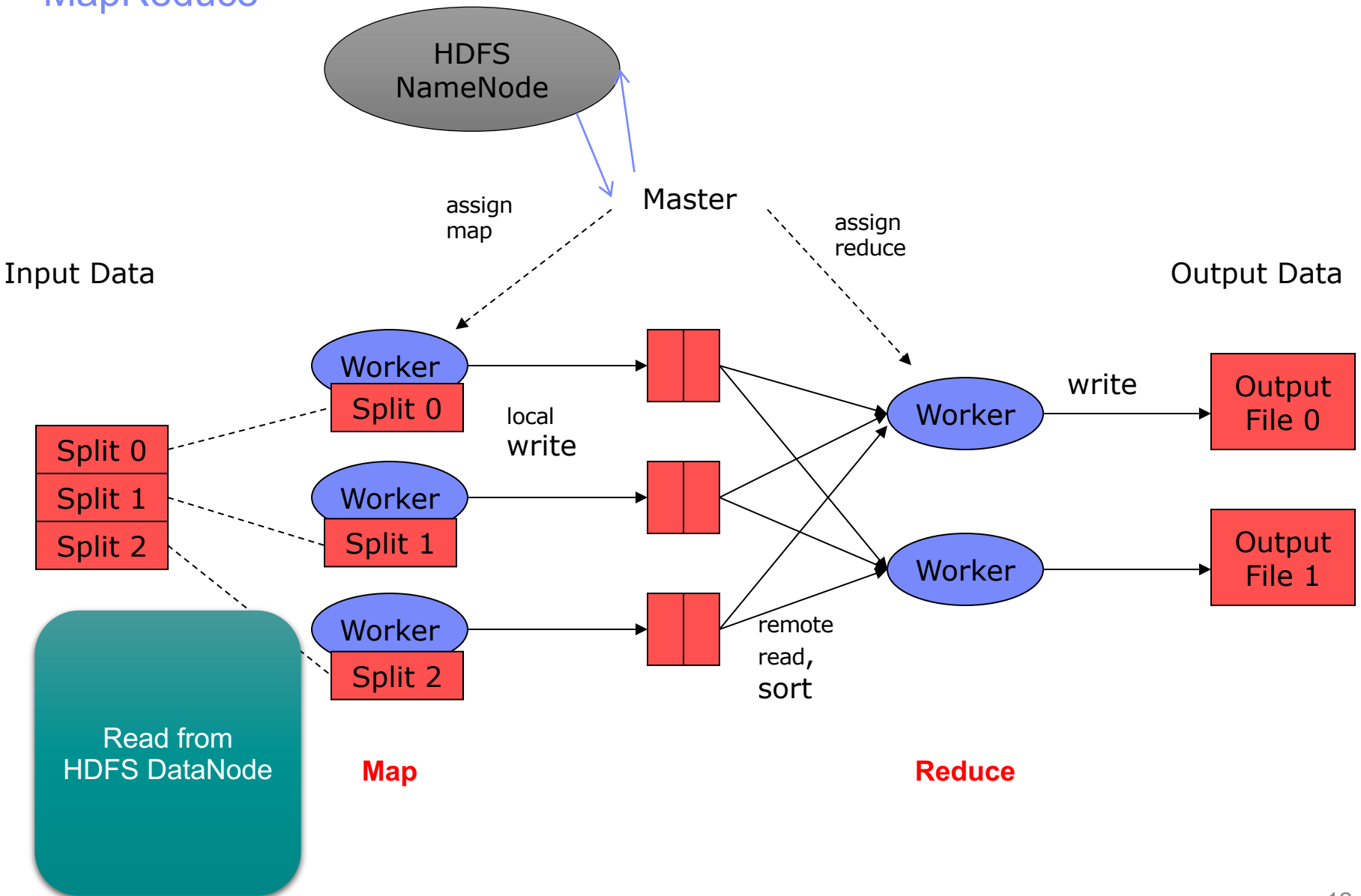
- Accepts the **Mapper output**, and aggregates values on the key
 - For our example, the reducer input would be:
<The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <**store**, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <**store**, 1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <**store**, 1> <opens, 1> <at, 1> <9am, 1>
 - The output would be:
<The, 6> <teacher, 1> <went, 1> <to, 1> <**store**, 3> <was, 1> <closed, 1> <opens, 1> <morning, 1> <at, 1> <9am, 1>

Another example: Chaining MapReduce

Count of URL access frequency

- Input: Log of accessed URLs, e.g., from proxy server
- Output: For each URL, % of total accesses for that URL
- First step:
 - Map – Process web log and outputs $\langle \text{URL}, 1 \rangle$
 - Multiple Reducers - Emits $\langle \text{URL}, \text{URL_count} \rangle$
(So far, like Wordcount. But still need %)
- Chain another MapReduce job after above one
 - Map – Processes $\langle \text{URL}, \text{URL_count} \rangle$ and outputs $\langle 1, (\langle \text{URL}, \text{URL_count} \rangle) \rangle$
 - 1 Reducer – Does two passes. In first pass, sums up all URL_count 's to calculate overall_count . In second pass calculates %'s
Emits multiple $\langle \text{URL}, \text{URL_count}/\text{overall_count} \rangle$

MapReduce



Locality Optimization

- **Master scheduling policy:**
 - Asks HDFS for locations of replicas of input file blocks
 - Map tasks scheduled so HDFS input block replica are on same machine or same rack
- **Effect:** Thousands of machines read input at local disk speed
 - Don't need to transfer input data all over the cluster over the network: eliminate network bottleneck!

Failure in MapReduce

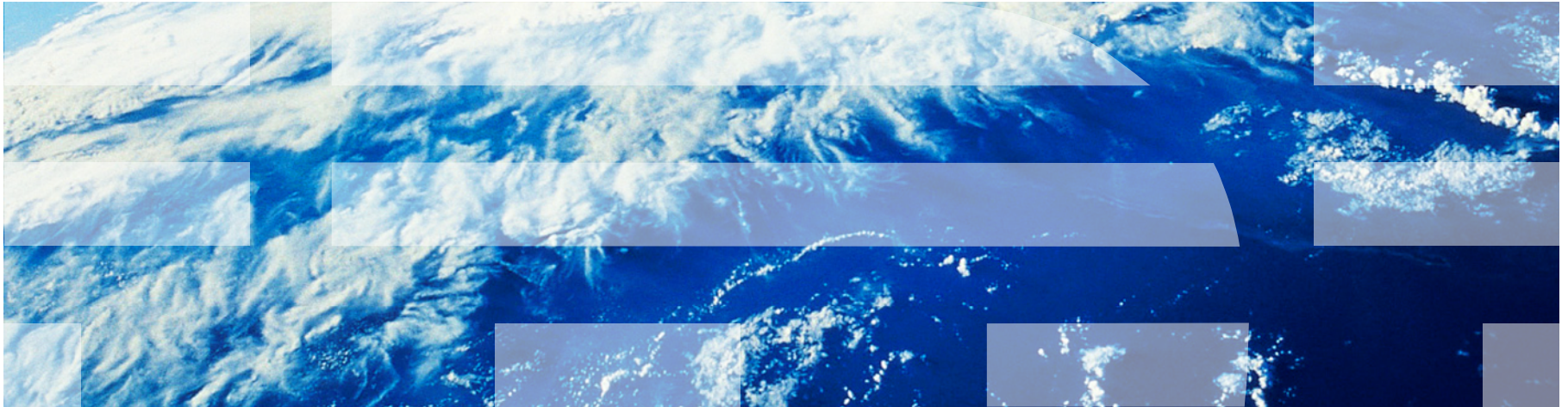
- Failures are the norm in commodity hardware
- **Worker failure**
 - Detect failure via periodic heartbeats
 - Re-execute in-progress map/reduce tasks
- **Master failure**
 - Single point of failure; Resume from Execution Log
- **Robust**
 - Google's experience: lost 1600 of 1800 machines once, but finished fine

Refinement: Redundant Execution

- Slow workers or **stragglers** significantly lengthen completion time
 - Slowest worker can determine the total latency!
 - This is why many systems measure **99th percentile latency**
 - Other jobs consuming resources on machine
 - Bad disks with soft errors transfer data very slowly
- **Solution:** spawn backup copies of tasks
 - Whichever one finishes first "wins"
 - I.e., treat slow executions as failed execute

Borrowed from Indranil Gupta, Faria Kalim, Patrick Wendell

Spark



Motivation

- Map reduce based tasks are slow
 - Data written to and read from storage
- Iterative algorithms not supported
 - Need to chain map reduce jobs → cumbersome, need to know how many jobs in advance (hard to do a loop)
- No support for interactive data mining and analytics

Key Concept: Resilient Distributed Datasets (RDDs)

- A form of **distributed shared memory**
 - Eliminates the need to read/write to/from disk intermediate data between iterations
 - Read only / immutable, partitioned collections of records
 - Deterministic
 - Formed by specific operations (map, filter, join, etc.)
 - Can be read from stable storage or other RDDs
 - User controlled persistence
 - User controlled partitioning
- More expressive interface than MapReduce
 - Transformations (e.g. map, filter, groupBy)
 - Actions (e.g. count, collect, save)
- You can even run SQL over Spark!

Spark programming interface

- Lazy operations
 - Transformations not done until action
- Operations on RDDs
 - Transformations - build new RDDs
 - Actions - compute and output results
- Partitioning – layout across nodes
- Persistence – final output can be stored on disk

RDD on Spark

| | |
|------------------------|---|
| Transformations | $\begin{aligned} \text{map}(f : T \Rightarrow U) &: \text{RDD}[T] \Rightarrow \text{RDD}[U] \\ \text{filter}(f : T \Rightarrow \text{Bool}) &: \text{RDD}[T] \Rightarrow \text{RDD}[T] \\ \text{flatMap}(f : T \Rightarrow \text{Seq}[U]) &: \text{RDD}[T] \Rightarrow \text{RDD}[U] \\ \text{sample}(\text{fraction} : \text{Float}) &: \text{RDD}[T] \Rightarrow \text{RDD}[T] \text{ (Deterministic sampling)} \\ \text{groupByKey}() &: \text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, \text{Seq}[V])] \\ \text{reduceByKey}(f : (V, V) \Rightarrow V) &: \text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)] \\ \text{union}() &: (\text{RDD}[T], \text{RDD}[T]) \Rightarrow \text{RDD}[T] \\ \text{join}() &: (\text{RDD}[(K, V)], \text{RDD}[(K, W)]) \Rightarrow \text{RDD}[(K, (V, W))] \\ \text{cogroup}() &: (\text{RDD}[(K, V)], \text{RDD}[(K, W)]) \Rightarrow \text{RDD}[(K, (\text{Seq}[V], \text{Seq}[W]))] \\ \text{crossProduct}() &: (\text{RDD}[T], \text{RDD}[U]) \Rightarrow \text{RDD}[(T, U)] \\ \text{mapValues}(f : V \Rightarrow W) &: \text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, W)] \text{ (Preserves partitioning)} \\ \text{sort}(c : \text{Comparator}[K]) &: \text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)] \\ \text{partitionBy}(p : \text{Partitioner}[K]) &: \text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)] \end{aligned}$ |
| Actions | $\begin{aligned} \text{count}() &: \text{RDD}[T] \Rightarrow \text{Long} \\ \text{collect}() &: \text{RDD}[T] \Rightarrow \text{Seq}[T] \\ \text{reduce}(f : (T, T) \Rightarrow T) &: \text{RDD}[T] \Rightarrow T \\ \text{lookup}(k : K) &: \text{RDD}[(K, V)] \Rightarrow \text{Seq}[V] \text{ (On hash/range partitioned RDDs)} \\ \text{save}(\text{path} : \text{String}) &: \text{Outputs RDD to a storage system, e.g., HDFS} \end{aligned}$ |

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

Transformed RDD

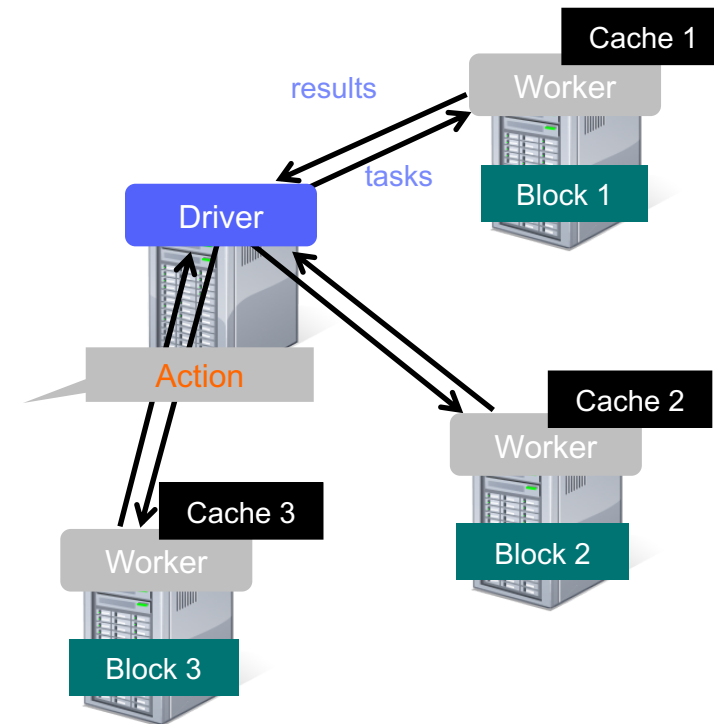
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```

...

Full-text search of Wikipedia

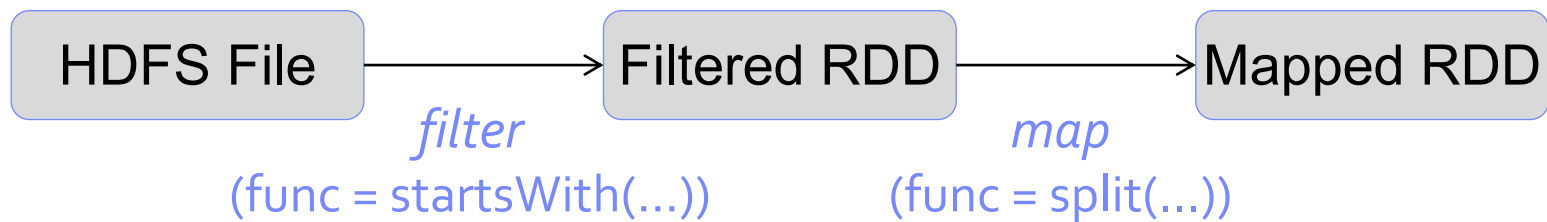
- 60GB on 20 EC2 machine
- 0.5 sec vs. 20s for on-disk



Fault Recovery

RDDs track **lineage** information that can be used to efficiently recompute lost data

```
msgs = textFile.filter(lambda s: s.startsWith("ERROR"))  
               .map(lambda s: s.split("\t")[2])
```



Creating RDDs

Turn a Python collection into an RDD

```
> sc.parallelize([1, 2, 3])
```

Load text file from local FS, HDFS, or S3

```
> sc.textFile("file.txt")
```

```
> sc.textFile("directory/*.txt")
```

```
> sc.textFile("hdfs://namenode:9000/path/file")
```

Basic Transformations

```
> nums = sc.parallelize([1, 2, 3])
```

```
# Pass each element through a function
```

```
> squares = nums.map(lambda x: x*x) // {1, 4, 9}
```

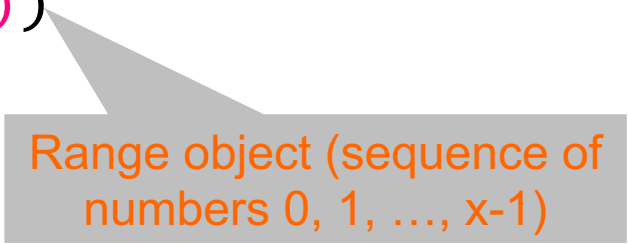
```
# Keep elements passing a predicate
```

```
> even = squares.filter(lambda x: x % 2 == 0) // {4}
```

```
# Map each element to zero or more others
```

```
> nums.flatMap(lambda x: => range(x))
```

```
> # => {0, 0, 1, 0, 1, 2}
```



Range object (sequence of numbers 0, 1, ..., x-1)

Basic Actions

```
> nums = sc.parallelize([1, 2, 3])  
  
# Retrieve RDD contents as a local collection  
> nums.collect() # => [1, 2, 3]  
  
# Return first K elements  
> nums.take(2)    # => [1, 2]  
  
# Count number of elements  
> nums.count()    # => 3  
  
# Merge elements with an associative function  
> nums.reduce(lambda x, y: x + y) # => 6  
  
# Write elements to a text file  
> nums.saveAsTextFile("hdfs://file.txt")
```

Working with Key-Value Pairs

Spark's “distributed reduce” transformations operate on RDDs of key-value pairs

Python:

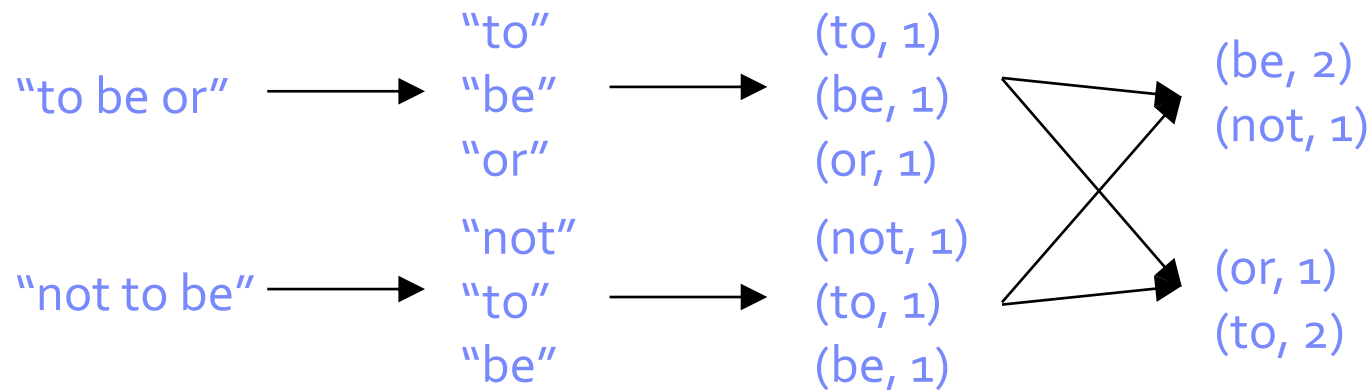
```
pair = (a, b)
pair[0] # => a
pair[1] # => b
```

Some Key-Value Operations

```
> pets = sc.parallelize(  
    [("cat", 1), ("dog", 1), ("cat", 2)])  
> pets.reduceByKey(lambda x, y: x + y)  
    # => {(cat, 3), (dog, 1)}  
> pets.groupByKey() # => {(cat, [1, 2]), (dog, [1])}  
> pets.sortByKey()  # => {(cat, 1), (cat, 2), (dog, 1)}
```

Example: Word Count

```
> lines = sc.textFile("hamlet.txt")
> counts = lines.flatMap(lambda line: line.split(" "))
                  .map(lambda word => (word, 1))
                  .reduceByKey(lambda x, y: x + y)
```

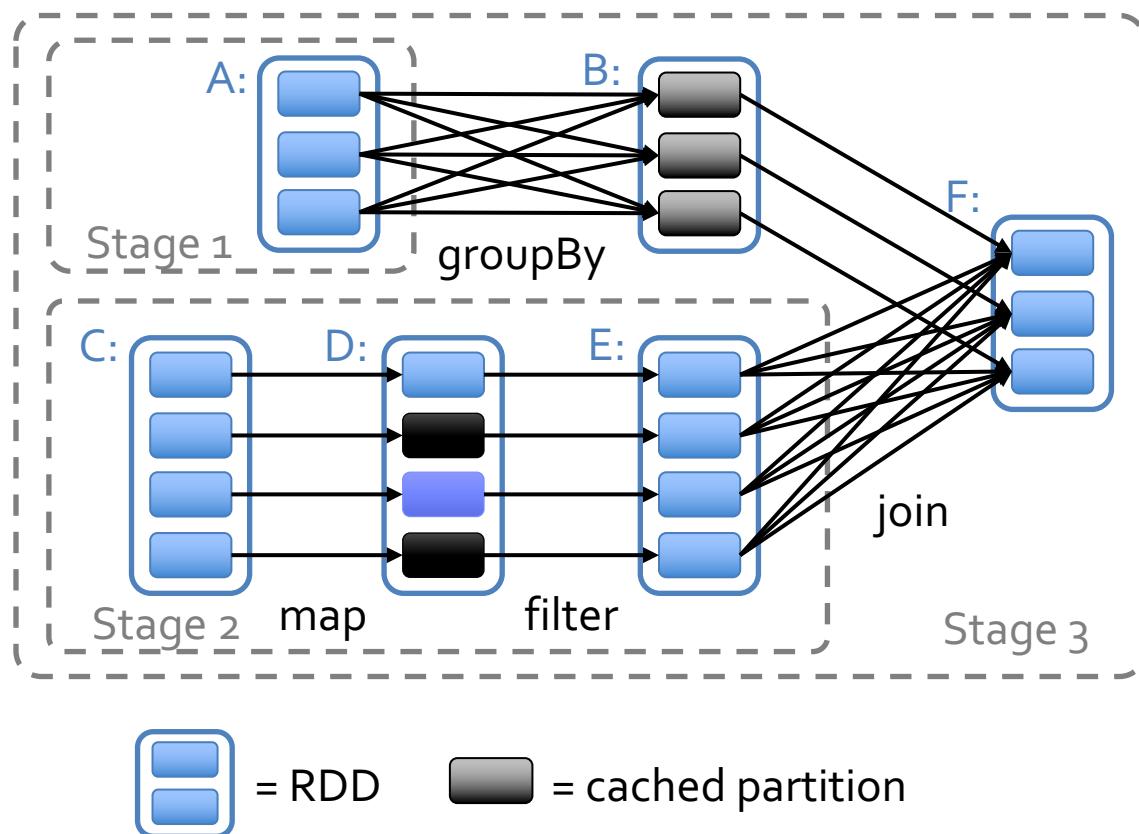


More RDD Operators

- | | | |
|------------------|---------------|-------------|
| ■ map | ■ reduce | sample |
| ■ filter | ■ count | take |
| ■ groupBy | ■ fold | first |
| ■ sort | ■ reduceByKey | partitionBy |
| ■ union | ■ groupByKey | mapWith |
| ■ join | ■ cogroup | pipe |
| ■ leftOuterJoin | ■ cross | save |
| ■ rightOuterJoin | ■ zip | ... |

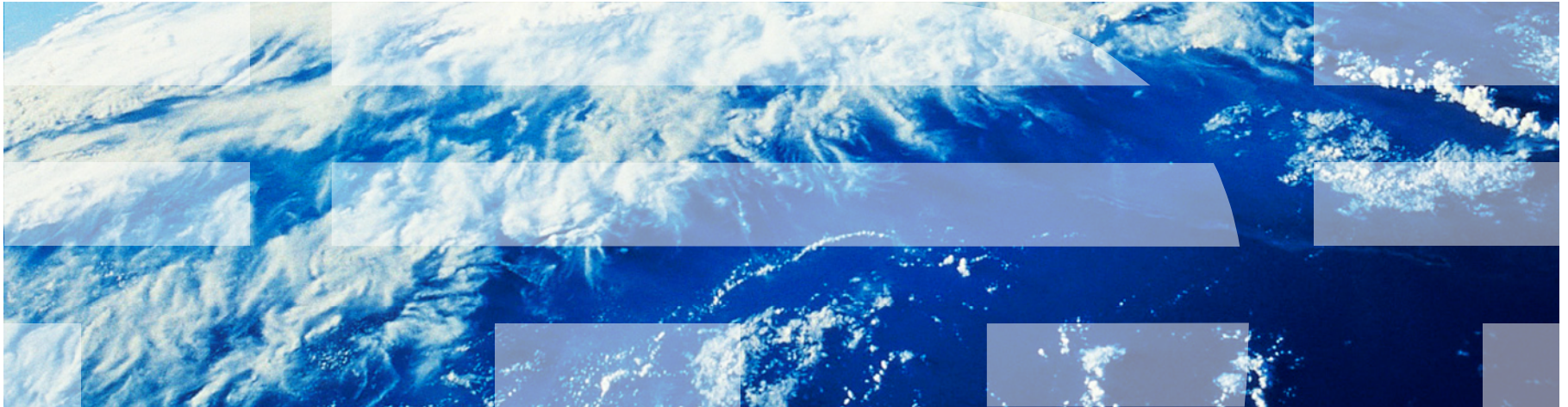
Under The Hood: DAG Scheduler

- General task graphs
- Automatically pipelines functions
- Data locality aware
- Partitioning aware to avoid shuffles



Borrowed from Tathagata Das and Indranil Gupta

Stream Processing



Motivation

- Large amounts of data => Need for real-time views of data
 - Social network trends, e.g., Twitter real-time search
 - Website statistics, e.g., Google Analytics
 - Intrusion detection systems, e.g., in most datacenters
- Process large amounts of data
 - With latencies of few seconds
 - With high throughput

Would MapReduce or normal Spark work?

- **Batch Processing** => need to wait for entire computation on large dataset to complete
- Not intended for long-running and real-time stream-processing

Which one of these is a stream processing job?

- A) Uber
Calculating surge prices
- B) LinkedIn
Aggregating updates into one email
- C) Netflix
Understanding user behavior to improve personalization
- D) TripAdvisor
Calculating earnings per day & fraud detection
- E) All of them

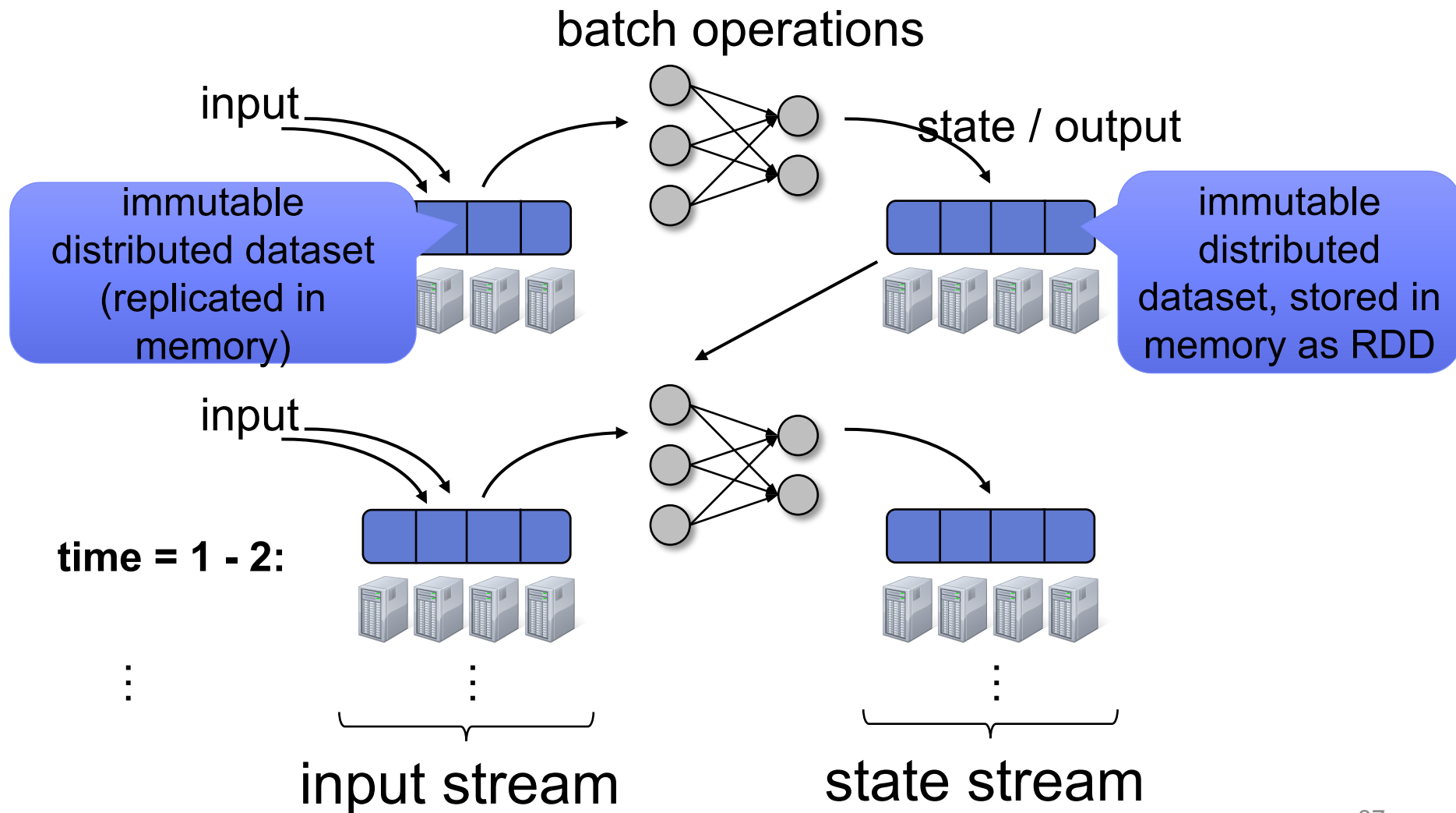
Discretized Stream Processing

- Run a streaming computation as a **series of very small, deterministic batch jobs**
- Batch processing models, like MapReduce, recover from faults and stragglers efficiently
 - Divide job into deterministic tasks
 - Rerun failed/slow tasks in parallel on other nodes
- Same recovery techniques at lower time scales

Spark Streaming

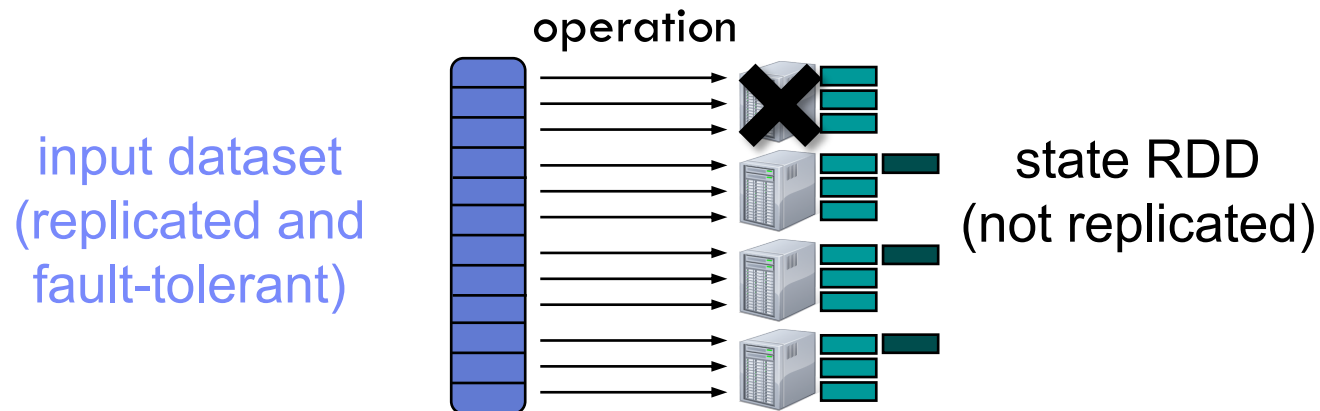
- State between batches kept in memory as **immutable, fault-tolerant dataset**
 - Specifically as Spark's Resilient Distributed Dataset
- Batch sizes can be reduced to as low as 1/2 second to achieve ~ 1 second latency
- Combines streaming and batch workloads
- Many other alternatives:
 - Apache Storm
 - Apache Flink
 - Amazon Kinesis
 - Google Dataflow
 - ...

Discretized Stream Processing



Fault Recovery

- State stored as RDD
 - Deterministically re-computable parallel collection
 - Remembers lineage of operations used to create them
- Fault / straggler recovery is done **in parallel** on other nodes



Fast recovery from faults
without full data replication

Programming Model

- A Discretized Stream or **DStream** is a series of RDDs representing a stream of data
 - API *very similar* to RDDs
- DStreams can be created...
 - Either from live streaming data
 - Or by transforming other DStreams

DStream Data Sources

- Many data sources can be inputs
 - HDFS
 - Kafka
 - Flume
 - Twitter
 - ...

Transformations

Build new streams from existing streams

- Filters/aggregate operations
 - map, flatMap, filter, count, reduce,
 - groupByKey, reduceByKey, sortByKey, join
 - etc.
- New window and stateful operations
 - window, countByWindow, reduceByWindow
 - countByValueAndWindow, reduceByKeyAndWindow
 - updateStateByKey
 - etc.

Output Operations

Send data to outside world

- `saveAsHadoopFiles`
- `print` – prints on the driver's screen
- `foreach` - arbitrary operation on every RDD

Example

Process a stream of Tweets to find the 20 most popular hashtags in the last 10 mins

1. Get the stream of Tweets and isolate the hashtags
2. Count the hashtags over 10 minute window
3. Sort the hashtags by their counts
4. Get the top 20 hashtags

1. Get the stream of Hashtags

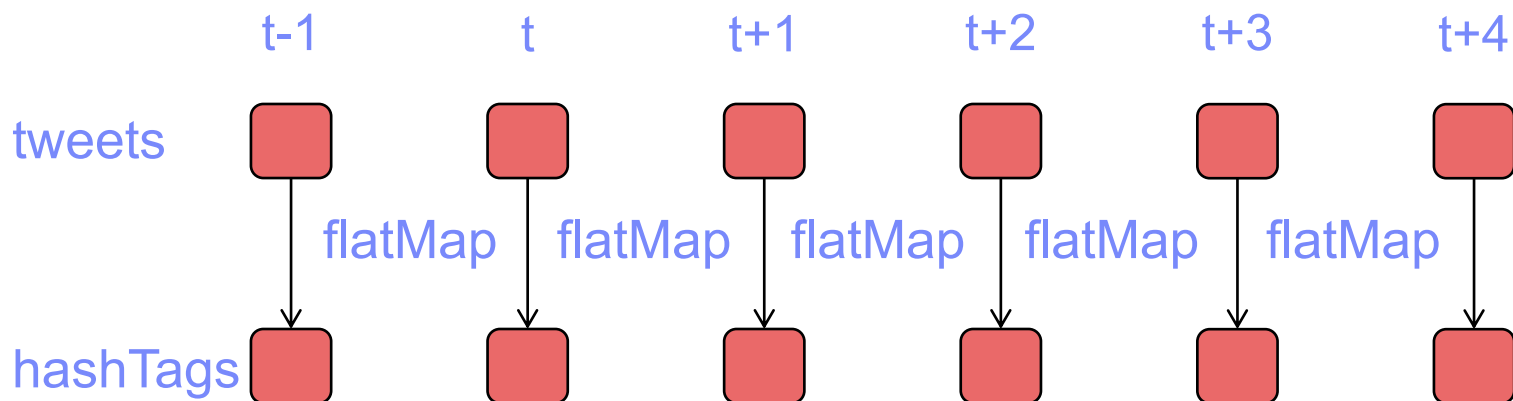
```
val tweets = ssc.twitterStream(<username>, <password>)
```

DStream

```
val hashtags = tweets.flatMap(status => getTags(status))
```

transformation

 = RDD

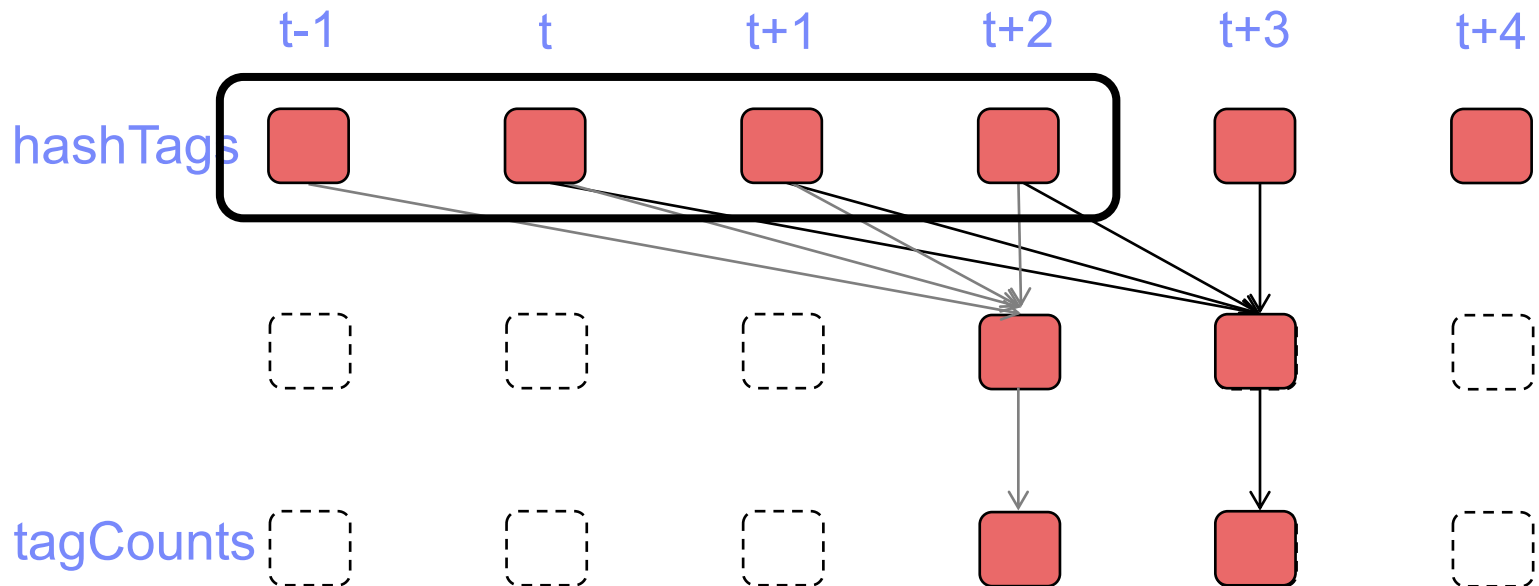


2. Count the hashtags over 10 min

```
val tweets = ssc.twitterStream(<div data-bbox="408 215 662 368" data-label="Text">

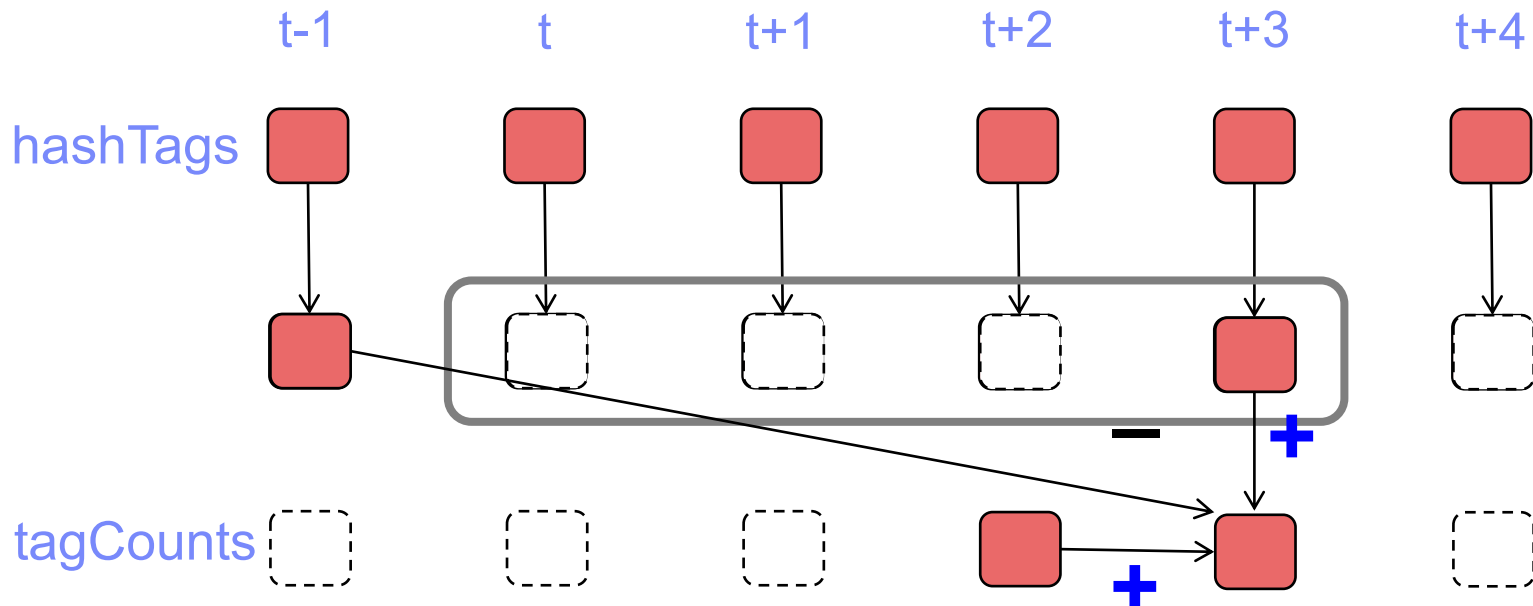
sliding window  
operation

val hashtags = tweets.flatMap (hashtag => hashtag.tags) (s))  
val tagCounts = hashtags.window(Minutes(10), Seconds(1))  
                .map(tag => (tag, 1)).reduceByKey(_ + _)
```



2. Count the hashtags over 10 min

```
val tweets = ssc.twitterStream(<username>, <password>)  
val hashtags = tweets.flatMap (status => getTags(status))  
val tagCounts = hashtags.countByValueAndWindow(Minutes(10), Seconds(1))
```



Smart window-based *reduce*

- Technique with count generalizes to reduce
 - Need a function to “subtract”
 - Applies to *invertible* reduce functions
- Could have implemented counting as:

```
hashTags.reduceByKeyAndWindow(_ + _, _ - _, Minutes(1), ...)
```

3. Sort the hashtags by their counts

```
val tweets = ssc.twitterStream(<username>, <password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashtags

                                .countByValueAndWindow(Minutes(1), Seconds(1))

val sortedTags = tagCounts.map { case (tag, cnt) => (cnt, tag) }
                        .transform(_._sortByKey(false))
```

allows arbitrary RDD
operations to create
a new DStream

4. Get the top 20 hashtags

```
val tweets = ssc.twitterStream(<username>, <password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashtags
                                .countByValueAndWindow(Minutes(1), Seconds(1))
val sortedTags = tagCounts.map { case (tag, cnt) => (cnt, tag) }
                                .transform(_._sortByKey(false))

sortedTags.foreach(showTopTags(20) _)
```



output operation